Bottom-up Reuse Guidelines: Hierarchy of Reuse

Bottom-up Reuse Guidelines: Hierarchy of Reuse

by Mark Sherman (SGT / NASA GSFC) with James Marshall (Innovim / NASA GSFC)

Based on the presentation, "Hierarchy of Reuse -- Designing Systems to Promote Reuse", by Mark Sherman, to the NASA 5th Joint Earth Science Data System Working Groups Meeting, Software Reuse Working Group Breakout Session, University of Maryland, Adelphi, MD, November 14, 2006.

Levels of Reuse

Reuse is possible at multiple levels. Going from smaller and easier to reuse to larger and harder to reuse, examples of these levels are:

- Function
- Class
- Function Library
- Class Library
- Subsystem
- System

The challenge is to make it easier to reuse at the subsystem and system levels.

Making Software Attractive for Reuse

For any software to be attractive for reuse, it must help a developer meet the following goals:

- Lower development costs
- Speed up deployment
- Reduce development risks

Attributes of software than make it attractive for reuse:

1. Source code available
2. Well documented with clear APIs
3. Fits the need without modification
4. Carries no excess baggage
5. Is thoroughly tested
6. Is in widespread use
7. Has broad community support

Attributes 1-5 are under the control of developers. It is hard to find software that fits the need without modification, but the less modifications required, the better.

Modular Designs

Hierarchical reuse uses modular designs: each layer fills a well-defined role and is fully documented and tested. This promotes reuse at multiple hierarchical levels, and if every level is documented and tested, it has reuse potential.

Maximizing Reuse Potential

A number of steps can be taken to help maximize the reuse potential for a system.

- Categorize and organize generic modules into reusable generic libraries
- Organize implementation-specific modules in separate libraries
- Build in layers on only the generic modules to create generic subsystems when possible
- Introduce specific modules as far up the hierarchy as possible to create implementation-specific subsystems and systems
- Document and test every module, subsystem, and system

Isolating the implementation-specific modules from each other and from the generic modules of a system is the key to designing for reuse. Such isolation limits the chain of dependencies for any module to strictly generic modules that are more readily reused than implementation specific modules.

Beyond the type of isolation described above, the ultimate in reuse potential is obtained by following the "dependency inversion principle". In essence, this principle states that no module, specific or generic, shall depend on anything other than a generic interface to another module. In other words, any module can be reused by implementing new modules for the generic interfaces on which the reused module depends.

Reuse often begins with making a system or its components reusable by the original developers, then reusable by others. From there, the sources are released to the software development community where external contributors improve the software. Code changes and other related information should be incorporated into configuration management. Changes should be fed back to the original developers and community so that others can benefit from them, and the changes should be as generic as possible. Any restrictions, etc. complicate matters.